IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	David C. Collins	Examiner:	Jeffrey S. Smith
Serial No.:	10/820,952	Group Art Unit:	2624
Filed:	April 8, 2004	Docket No.:	200400670-1
Title:	GENERATING AND DISPLAYING SPATIALLY OFFSET SUB-FRAMES		

DECLARATION OF PRIOR INVENTION UNDER 37 C.F.R. § 1.131

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir/Madam:

This Declaration is submitted to establish prior invention of the subject matter of the present patent application. The person making this Declaration is inventor David C. Collins.

Accompanying this Declaration is Exhibit A to establish reduction to practice of the subject matter of the present patent application in the United States prior to the publication date of February 12, 2004 of U.S. Patent Application No. US 2004/0027363 (hereinafter referred to as "Allen").

Exhibit A (14 pages) includes a Hewlett-Packard Company (HP) Invention Disclosure and attachment submitted by the inventor and received by the HP Legal Department prior to February 12, 2004. In addition, this invention disclosure was witnessed prior to February 12, 2004. This invention disclosure was assigned HP Patent Disclosure No. 200400670. The invention disclosure and attachment describe the subject matter of the present patent application.

From these exhibits, it can be seen that the subject matter of the present patent application was reduced to practice prior to the publication date of February 12, 2004 of Allen.

As a person signing below, I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Signed: David C. Collins
David C. Collins

Date: 28-May-2008

200400670: A Practical Implementati... [redacted]

Innovation Number
200400670 [redacted]

Disclosure Summary

Title A Practical Implementation of the 2 Position Multipass Center Adaptive Algorithm

Abstract The 2 position center adaptive algorithm cannot be similified in the same fashion as the 4 position center adaptive algorithm. This adds a complication to the actual implementation of the algorithm in the ASIC. This disclosure shows how to overcome this complication. This is done by storing both the final subframe values and the error values from the previous row.

Attachments Adaptive_Kernel_With_History_2Pos.doc [285696 bytes] [redacted]

Inventors David C Collins

Invention Disclosures

Disclosure No. 200400670

[redacted]

PD No.	Date/Time Submitted	Collection
200400670	[redacted]	

General Information

Title	A Practical Implementation of the 2 Position Multipass Center Adaptive Algorithm
Abstract	The 2 position center adaptive algorithm cannot be similified in the same fashion as the 4 position center adaptive algorithm. This adds a complication to the actual implementation of the algorithm in the ASIC. This disclosure shows how to overcome this complication. This is done by storing both the final subframe values and the error values from the previous row.
Projects	[redacted]
Products	[redacted]

Attachments

Attachments	Adaptive_Kernel_With_History_2Pos.doc [285696 bytes] - [redacted]

Description of Invention

Problems Solved	This invention shows how to implement a mutlipass center adaptive algorithm for 2 position wobulation using the same memory requirements as the simplified 4 position center adaptive algorithm
Prior Solutions	Most of the prior solutions for 2 position wobulation have been single pass. My recent disclosure "A Practical Implementation of Multipass Adaptive Wobultion" can only be extended to the 2 position algorithm if the standard adaptive method is used, but the standard adaptive algorithm does not perform well for some types of input.
Description	The complete description is contained in the attatched document. For the implementation of the multipass adaptive algorithm, one region of memory is required to store the previously processed subframe row. The novelty of this invention is storing the error values from the previous processed row as well. In particular the error value that is stored is as follows: error = error_left + 2*error + error_right By storing the error values in this fashion, the error values and the final subframe values can both be stored in an interleaved fashion in the same region of memory. This enables the 2 position algorithm and the 4 position algorithm to be implemented using the same amount of memory. The other novelty of this invention is transforming the error value from a signed number containing many bits to an 8 bit number. A simple conversion routine is defined in the attatched document, and another embodiment would be to use a look up table to do the conversion.
Advantages	The primary adavantage of this algorithm is that it enables to 2 position center adaptive wobulation algorithm to be performed using the same memory requirements as the 4 position algorithm. This invention enables one ASIC to be development that contains both algorithms with out any unnecessary memory. Without this invention the 3 pass 2 position center adaptive algorithm would require and additional 20% of on chip memory.

Invention History

Published	[redacted]
Announced	
Disclosed	
Next Three Months	
Described	
Built	
Government Contract	
Related Disclosure	

Innovation Workshop	No

Inventor Information

Inventors	David C Collins Hewlett-Packard Company Corvallis [redacted]

Witnesses

Witnesses	Matthew P Heineck Hewlett-Packard Company Corvallis [redacted]

Classification

Recommended Classification	[redacted]
Keywords	
Recommended Merlin Entity	
Recommended Merlin Loc	
Recommended Merlin Responsible_attorney	

Administrative Record

Date/Time Submitted	[redacted]
PD Number	200400670
Date PD Number Assigned	[redacted]

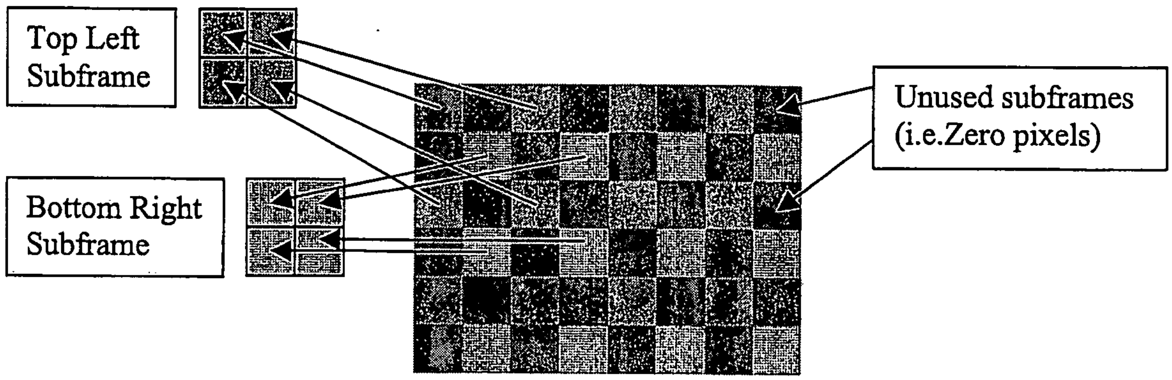
Filing Decisions

[redacted]

Adaptive Kernel With History – 2 Position

Background

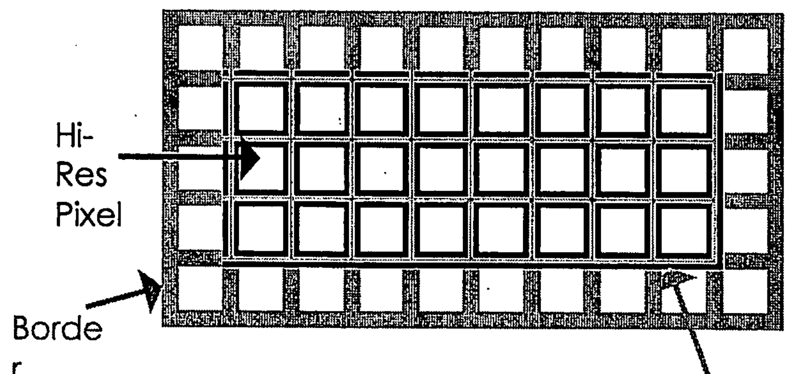
The following explanation is assuming 2 position wobulation. Thus, two low resolution subframes must be generated for each frame – one for each wobulation position. Both of the subframes are processed together at the same time, and the subframes are intertwined. Thus, every second pixel will correspond to a different subframe. The following diagram illustrates the idea. The grey pixels are not used for two position wobulation.



To compute the optimal solution, and iterative algorithm can be used. The iterative approach has been called the adaptive algorithm. The standard 4 position adaptive algorithm is described below.

Simulation Kernel

$$(K_s)\quad \begin{matrix} \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 0 \end{matrix}$$



Error Kernel

$$(K_e)\quad \begin{matrix} 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{4} \end{matrix}$$

Iterative Algorithm

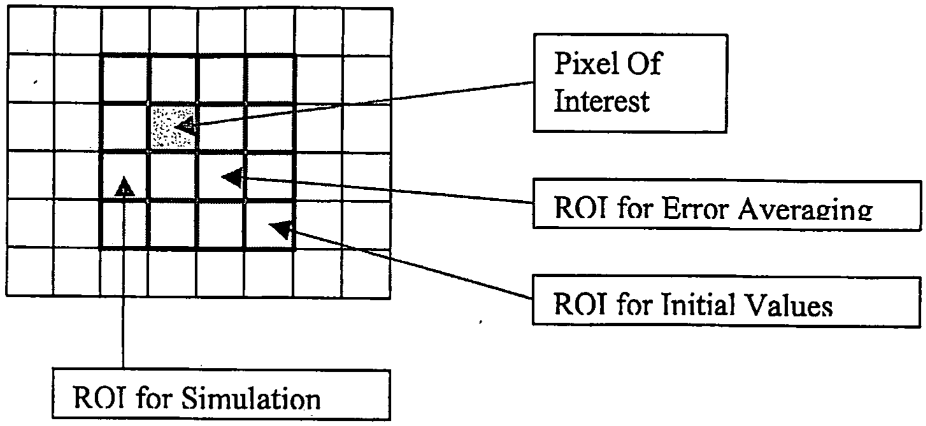
Simulation = K_s * Subframes

Error = Image – Simulation

$Error_{avg}$ = K_e * Error

Subframes = Subframes + α x $Error_{avg}$

In a previous disclosure, I showed how the 4 position standard adaptive algorithm could be implemented in one pass using a small region of interest. One of the key discoveries was that the pixels above the pixel of interest were only needed to generate a simulated image (all the subframes merge together). The following diagram illustrates the region of interest for one pass of the 4 position standard adaptive algorithm.



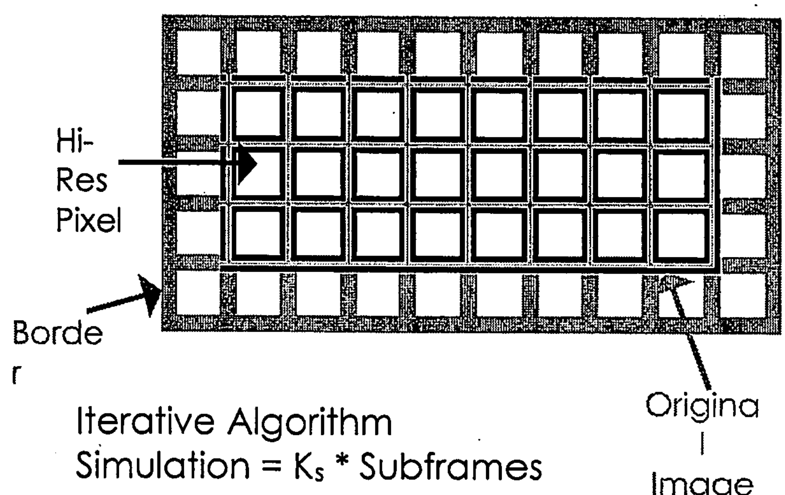
Since the pixels above the pixel of interest are only required for the simulation ROI, the calculated values for the previous row are exactly the values that are needed. This same approach works for the 2 position standard adaptive method. The only change is that the simulation kernel has a value of ½ instead of ¼. This change occurs because only half of the high resolution pixels are non-zero. Everything else for the adaptive kernel with history works exactly as was disclosed before.

Simulation Kernel

$$(K_s)\quad \begin{matrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{matrix}$$

Error Kernel

$$(K_e)\quad \begin{matrix} 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{4} \end{matrix}$$



Iterative Algorithm

Simulation = K_s * Subframes

Error = Image – Simulation

$Error_{avg}$ = K_e * Error

Subframes = Subframes + α x $Error_{avg}$

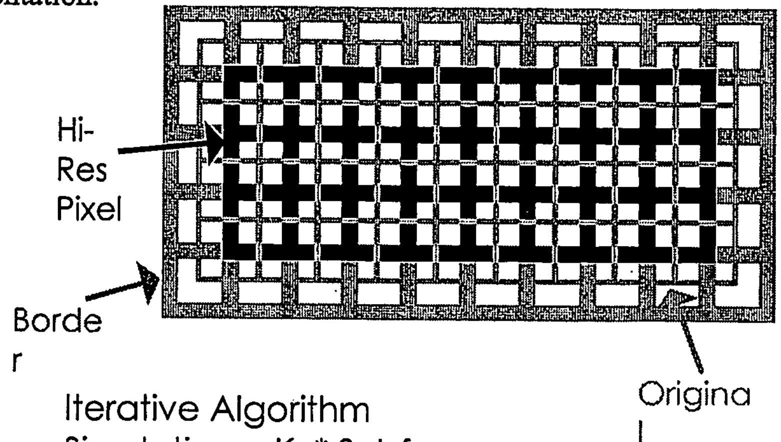
The next algorithm that I considered for 4 position wobulation was the center adaptive algorithm. This algorithm performed better than the standard adaptive algorithm on single pixel lines. The center adaptive algorithm has several drawbacks in terms of implementation. First it includes many more computations than the standard adaptive algorithm. The second disadvantage is that both the error ROI and the simulation ROI for a given pixel extend above and below the pixel of interest. This implies that for the adaptive kernel with history. Both the final subframe values and error values from the

previous row are required for the algorithm. Thus more on chip memory would be required for an ASIC implementation.

Simulation Kernel

(K_s)

$$\begin{matrix} \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{matrix}$$



Error Kernel

(K_e)

$$\begin{matrix} \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{matrix}$$

Iterative Algorithm

Simulation = K_s * Subframes

Error = Image – Simulation

$Error_{avg}$ = K_e * Error

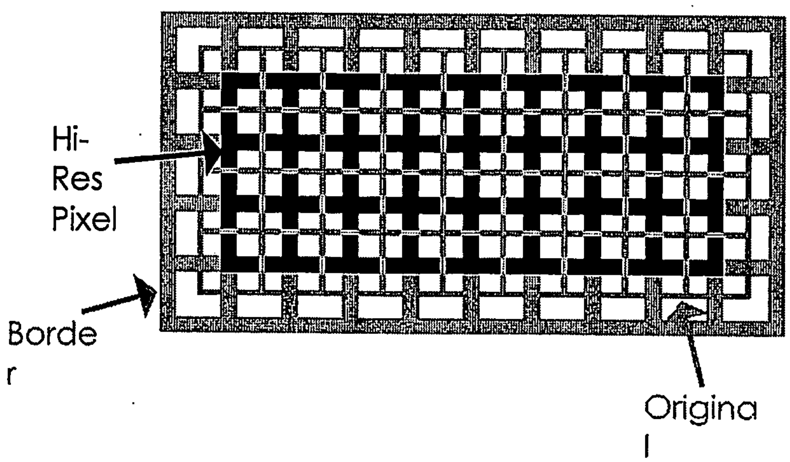
Subframes = Subframes + α x $Error_{avg}$

To overcome the implementation difficulties with the center adaptive algorithm. I presented a simplified center adaptive algorithm. This simplified center algorithm performed comparably to the center adaptive algorithm, and it was less computationally expensive than even that standard adaptive algorithm. The simplified algorithm is given below. One of the key features to note is that the error averaging step has been eliminated. This is what allowed the adaptive kernel with history approach to work without the burden of additional memory.

Simulation Kernel

(K_s)

$$\begin{matrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{4}{8} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{matrix}$$



Iterative Algorithm

Simulation = K_s * Subframes

Error = Image – Simulation

Subframes = Subframes + α x Error

Unfortunately these simplifications do not give satisfying results for two position wobulation. This approach fails because only half of the subframe values are non-zero. Two of the subframes don't exist at all, and this is the same as setting all their values to zero.

2 Position Center Adaptive Kernel with History

Consider again the complete center adaptive algorithm. For 2 position wobulation, conceptually, half of the high resolution values are zero (two of the four subframes don't exist at all). Thus, the simulation kernel can be reduced to the following two kernels. One kernel is used when the center hi-res pixel is non-zero, and the other kernel is used when the center pixel is zero. In addition, the error only needs to be averaged for the nonzero pixel location (the error only has to be feedback for subframes that exist).

Simulation Kernel

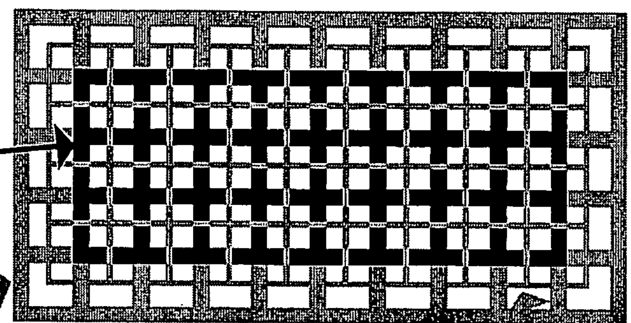
(K_s)

Nonzero Center Pixel

$$\begin{matrix} \frac{1}{8} & 0 & \frac{1}{8} \\ 0 & \frac{4}{8} & 0 \\ \frac{1}{8} & 0 & \frac{1}{8} \end{matrix}$$

Zero Center Pixel

$$\begin{matrix} 0 & \frac{2}{8} & 0 \\ \frac{2}{8} & 0 & \frac{2}{8} \\ 0 & \frac{2}{8} & 0 \end{matrix}$$



Error Kernel

(K_e)

Nonzero Center Pixel

$$\begin{matrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{matrix}$$

Iterative Algorithm

Simulation = K_s * Subframes

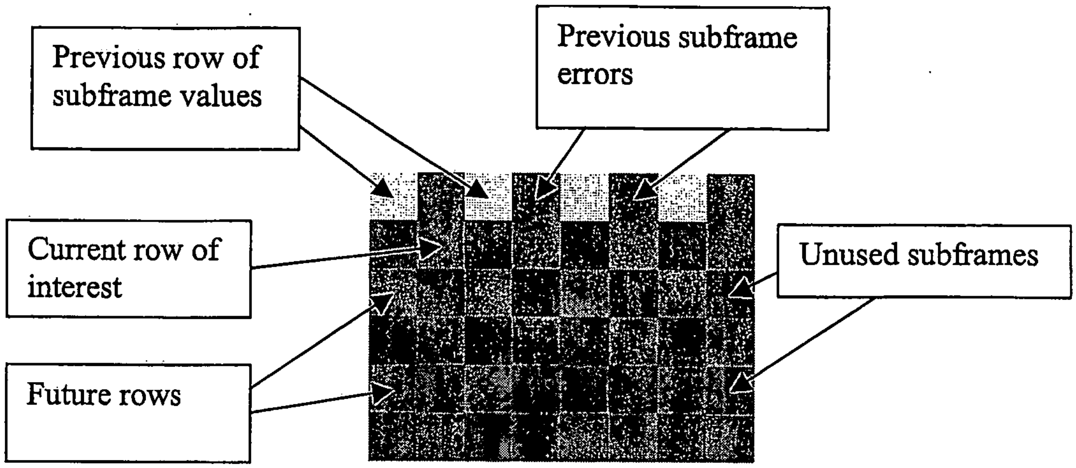
Error = Image – Simulation

$Error_{avg}$ = K_e * Error

Subframes = Subframes + α x $Error_{avg}$

Thus, the computational complexity is only half of the complete 4 position center adaptive algorithm. The obstacle of the error from the previous row still remains. The solution is to store both the final subframe values and the error values in one row of image memory. For a given high resolution row, only half of the locations are used to store subframe values. The other half of the values are unused or set to zero. This leaves half of the values for storing the errors.

The second observation is that for a given row in the hi-res image, only half of the values require the error to be averaged. The following diagram is an attempt to illustrate this point.



The pink error values actual contains a summation of the error value of the left and right pixels in addition to its own error value. In particular, each error value adheres to the following formula:

$$\text{Error} = 1{*}\text{error}_{\text{left\_pixel}} + 2{*}\text{error} + 1{*}\text{error}_{\text{right\_pixel}}$$

This matches the first row of the error averaging kernel. Thus, in one row of memory we can store both the required error values and the subframe values. The last little detail is that the error value is a signed value, and it contains more bits than a single pixel. To accommodate this we can use a lookup table or a simple mapping. Psuedo code for one such simple mapping is as follows:

```
temp = error_left+2*error+error_right; // 1x 2x 1x
temp = temp/4;                          // divide by 4
if( temp < -127 ) temp = -127;          // clip value
if( temp > 127  ) temp =  127;          // clip value
temp += 127;                            // shift to make non-zero
```

Thus, the key features to implement the 2 position center adaptive algorithm using the adaptive kernel with history is that both the final subframe values and the previous error values can be stored in one row of image memory. Thus, a 3 pass 2 position center adaptive algorithm can still be implemented with 1 row of history and 4 rows of image data.

Appendix

For completeness I have include some C++ code for calculating the center adaptive kernel with history. One thing to note is that I need to keep track of whether I am on an odd or an even pixel because I have to keep track of which subframe values are unused.

```
unsigned char AdaptiveCenterKernel_2Pos::Calculate
(       unsigned char final0,
        unsigned char image1,
        unsigned char image2,
        unsigned char image3,
        unsigned char image4,
        unsigned char isOddPixel
)
{
        int temp;

        // isOddPixel = true
        //    0    f0_1     0    f0_3     0    f0_5
        // g1_0     0    g1_2     0    g1_4     0
        //    0    g2_1     0    g2_3     0    g2_5
        // g3_0     0    g3_2     0    g3_4     0
        //    0    g4_1     0    g4_3     0    g4_5

        // isOddPixel = false
        // f0_0     0    f0_2     0    f0_4     0
        //    0    g1_1     0    g1_3     0    g1_5
        // g2_0     0    g2_2     0    g2_4     0
        //    0    g3_1     0    g3_3     0    g3_5
        // g4_0     0    g4_2     0    g4_4     0

        shiftValues();

        final0_5M = final0;
        image1_5M = image1;
        image2_5M = image2;
        image3_5M = image3;
        image4_5M = image4;

        // calculate guess for column 4
        // using pixel selection - no calculations here
        guess1_4M = image1_4M;
        guess2_4M = image2_4M;
        guess3_4M = image3_4M;
        guess4_4M = image4_4M;

        // compute sim column 4
        int sim1_4 = 0;
        int sim2_4 = 0;
        int sim3_4 = 0;
        int sim4_4 = 0;

        if( isOddPixel )
        {
                sim1_4 = final0_3M+guess2_3M+final0_5M+image2_5M;
                sim1_4 += guess1_4M<<2; // multiply by four

                sim2_4 = (guess1_4M<<1)+(guess2_3M<<1)+(image2_5M<<1)+(guess3_4M<<1);

                sim3_4 = guess2_3M+guess4_3M+image2_5M+image4_5M;
                sim3_4 += guess3_4M<<2; // multiply by four
```

```
        //sim4_4 = (guess3_4M<<1)+(guess4_3M<<1)+(guess4_5M<<1)+(guess5_4M<<1); invalid
        sim4_4 = (guess3_4M<<1)+(guess4_3M<<1)+(image4_5M<<1)+(guess4_4M<<1);
}
else
{
        sim1_4 = (final0_4M<<1)+(guess1_3M<<1)+(image1_5M<<1)+(guess2_4M<<1);

        sim2_4 = guess1_3M+guess3_3M+image1_5M+image3_5M;
        sim2_4 += guess2_4M<<2; // multiply by four

        sim3_4 = (guess2_4M<<1)+(guess3_3M<<1)+(image3_5M<<1)+(guess4_4M<<1);

        // sim4_4 = guess3_3M+guess5_3M+guess3_5M+guess5_5M; invalid
        sim4_4 = guess3_3M+guess4_3M+image3_5M+image4_5M;
        sim4_4 += guess4_4M<<2; // multiply by four
}

int err1_4  = (image1_4M<<3) - sim1_4;  // column 4
int err2_4  = (image2_4M<<3) - sim2_4;
int err3_4  = (image3_4M<<3) - sim3_4;
int err4_4  = (image4_4M<<3) - sim4_4;

// compute sim column 3
int sim1_3 = 0; // column 3
int sim2_3 = 0;
int sim3_3 = 0;
int sim4_3 = 0;

if( !isOddPixel )
{
        sim1_3 = final0_2M+guess2_2M+final0_4M+guess2_4M;
        sim1_3 += guess1_3M<<2; // multiply by four

        sim2_3 = (guess1_3M<<1)+(guess2_2M<<1)+(guess2_4M<<1)+(guess3_3M<<1);

        sim3_3 = guess2_2M+guess4_2M+guess2_4M+guess4_4M;
        sim3_3 += guess3_3M<<2; // multiply by four

        sim4_3 = (guess3_3M<<1)+(guess4_2M<<1)+(guess4_4M<<1)+(guess4_3M<<1);
}
else
{
        sim1_3 = (final0_3M<<1)+(guess1_2M<<1)+(guess1_4M<<1)+(guess2_3M<<1);

        sim2_3 = guess1_2M+guess3_2M+guess1_4M+guess3_4M;
        sim2_3 += guess2_3M<<2; // multiply by four

        sim3_3 = (guess2_3M<<1)+(guess3_2M<<1)+(guess3_4M<<1)+(guess4_3M<<1);

        // sim4_3 = guess3_2M+guess5_2M+guess3_4M+guess5_4M; invalid
        sim4_3 = guess3_2M+guess4_2M+guess3_4M+guess4_4M;
        sim4_3 += guess4_3M<<2; // multiply by four
}

int err1_3  = (image1_3M<<3) - sim1_3;  // column 3
int err2_3  = (image2_3M<<3) - sim2_3;
int err3_3  = (image3_3M<<3) - sim3_3;
int err4_3  = (image4_3M<<3) - sim4_3;

int sim1_2 = 0; // column 2
int sim2_2 = 0;
int sim3_2 = 0;
int sim4_2 = 0;
```

```
if( isOddPixel )
{
      sim1_2 = final0_1M+guess2_1M+final0_3M+guess2_3M;
      sim1_2 += guess1_2M<<2; // multiply by four

      sim2_2 = (guess1_2M<<1)+(guess2_1M<<1)+(guess2_3M<<1)+(guess3_2M<<1);

      sim3_2 = guess2_1M+guess4_1M+guess2_3M+guess4_3M;
      sim3_2 += guess3_2M<<2; // multiply by four

      //sim4_2 = (guess3_2M<<1)+(guess4_1M<<1)+(guess4_3M<<1)+(guess5_2M<<1); invalid
      sim4_2 = (guess3_2M<<1)+(guess4_1M<<1)+(guess4_3M<<1)+(guess4_2M<<1);
}
else
{
      sim1_2 = (final0_2M<<1)+(guess1_1M<<1)+(guess1_3M<<1)+(guess2_2M<<1);

      sim2_2 = guess1_1M+guess3_1M+guess1_3M+guess3_3M;
      sim2_2 += guess2_2M<<2; // multiply by four

      sim3_2 = (guess2_2M<<1)+(guess3_1M<<1)+(guess3_3M<<1)+(guess4_2M<<1);

      // sim4_2 = guess3_1M+guess5_1M+guess3_3M+guess5_3M; invalid
      sim4_2 = guess3_1M+guess4_1M+guess3_3M+guess4_3M;
      sim4_2 += guess4_2M<<2; // multiply by four
}

int err1_2  = (image1_2M<<3) - sim1_2;  // column 2
int err2_2  = (image2_2M<<3) - sim2_2;
int err3_2  = (image3_2M<<3) - sim3_2;
int err4_2  = (image4_2M<<3) - sim4_2;

// compute guess column 3
if( isOddPixel )
{
      temp  = (err1_2>>2) + (err1_3>>1) + (err1_4>>2);  // 2x 4x 2x
      temp += (err2_2>>1) +  err2_3     + (err2_4>>1);  // 4x 8x 4x
      temp += (err3_2>>2) + (err3_3>>1) + (err3_4>>2);  // 2x 4x 2x

      temp  = temp * alpha1M;
      temp  = temp >> 7;           // numerator is assumed to be 4
      temp  = temp + guess2_3M;
      guess2_3M = max(0,min(temp,255));       // clip value

      // err4_3 = don't update this value - we don't realy have enough information
      // I will need to check if an aproximation is better than nothing though
}
else
{
      //temp   = (err0_2>>2) + (err0_3>>1) + (err0_4>>2);  // 2x 4x 2x
      temp   = (final0_3M-127)<<3;  // 2x 4x 2x
      temp  += (err1_2>>1) +  err1_3     + (err1_4>>1);  // 4x 8x 4x
      temp  += (err2_2>>2) + (err2_3>>1) + (err2_4>>2);  // 2x 4x 2x

      temp  = temp * alpha1M;
      temp  = temp >> 7;           // numerator is assumed to be 4
      temp  = temp + guess1_3M;
      guess1_3M = max(0,min(temp,255));       // clip value

      temp   = (err2_2>>2) + (err2_3>>1) + (err2_4>>2);  // 2x 4x 2x
      temp  += (err3_2>>1) +  err3_3     + (err3_4>>1);  // 4x 8x 4x
      temp  += (err4_2>>2) + (err4_3>>1) + (err4_4>>2);  // 2x 4x 2x
```

```
        temp  = temp * alpha1M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess3_3M;
        guess3_3M = max(0,min(temp,255));      // clip value
}

// compute sim column 1
int sim1_1 = 0; // column 1
int sim2_1 = 0;
int sim3_1 = 0;
int sim4_1 = 0;

if( !isOddPixel )
{
        sim1_1 = final0_0M+guess2_0M+final0_2M+guess2_2M;
        sim1_1 += guess1_1M<<2; // multiply by four

        sim2_1 = (guess1_1M<<1)+(guess2_0M<<1)+(guess2_2M<<1)+(guess3_1M<<1);

        sim3_1 = guess2_0M+guess4_0M+guess2_2M+guess4_2M;
        sim3_1 += guess3_1M<<2; // multiply by four

        sim4_1 = (guess3_1M<<1)+(guess4_0M<<1)+(guess4_2M<<1)+(guess4_1M<<1);
}
else
{
        sim1_1 = (final0_1M<<1)+(guess1_0M<<1)+(guess1_2M<<1)+(guess2_1M<<1);

        sim2_1 = guess1_0M+guess3_0M+guess1_2M+guess3_2M;
        sim2_1 += guess2_1M<<2; // multiply by four

        sim3_1 = (guess2_1M<<1)+(guess3_0M<<1)+(guess3_2M<<1)+(guess4_1M<<1);

        sim4_1 = guess3_0M+guess4_0M+guess3_2M+guess4_2M;
        sim4_1 += guess4_1M<<2; // multiply by four
}

int err1_1  = (image1_1M<<3) - sim1_1;  // column 1
int err2_1  = (image2_1M<<3) - sim2_1;
int err3_1  = (image3_1M<<3) - sim3_1;
int err4_1  = (image4_1M<<3) - sim4_1;

// Compute Guess for Column 2
if( !isOddPixel )
{
        temp  = (err1_1>>2) + (err1_2>>1) + (err1_3>>2);  // 2x 4x 2x
        temp += (err2_1>>1) +  err2_2     + (err2_3>>1);  // 4x 8x 4x
        temp += (err3_1>>2) + (err3_2>>1) + (err3_3>>2);  // 2x 4x 2x

        temp  = temp * alpha2M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess2_2M;
        guess2_2M = max(0,min(temp,255));      // clip value

        // err4_3 = don't update this value - we don't realy have enough information
        // I will need to check if an aproximation is better than nothing though
}
else
{
        // temp   = (err0_1>>2) + (err0_2>>1) + (err0_3>>2);  // 2x 4x 2x
        temp   = (final0_2M-127)<<3;  // 2x 4x 2x
        temp  += (err1_1>>1) +  err1_2     + (err1_3>>1);  // 4x 8x 4x
```

```
        temp  += (err2_1>>2) + (err2_2>>1) + (err2_3>>2);  // 2x 4x 2x

        temp  = temp * alpha2M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess1_2M;
        guess1_2M = max(0,min(temp,255));      // clip value

        temp   = (err2_1>>2) + (err2_2>>1) + (err2_3>>2);  // 2x 4x 2x
        temp  += (err3_1>>1) +  err3_2     + (err3_3>>1);  // 4x 8x 4x
        temp  += (err4_1>>2) + (err4_2>>1) + (err4_3>>2);  // 2x 4x 2x

        temp  = temp * alpha2M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess3_2M;
        guess3_2M = max(0,min(temp,255));      // clip value
}

// Compute Guess for Column 1
if( isOddPixel )
{
        temp  = (err1_0M>>2) + (err1_1>>1) + (err1_2>>2);  // 2x 4x 2x
        temp += (err2_0M>>1) +  err2_1     + (err2_2>>1);  // 4x 8x 4x
        temp += (err3_0M>>2) + (err3_1>>1) + (err3_2>>2);  // 2x 4x 2x

        temp  = temp * alpha3M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess2_1M;
        guess2_1M = max(0,min(temp,255));      // clip value

        // err4_1 = don't update this value - we don't realy have enough information
        // I will need to check if an aproximation is better than nothing though
}
else
{
        //temp   = (err0_0M>>2) + (err0_1>>1) + (err0_2>>2);  // 2x 4x 2x
        temp   = (final0_1M-127)<<3;  // 2x 4x 2x
        temp  += (err1_0M>>1) +  err1_1     + (err1_2>>1);  // 4x 8x 4x
        temp  += (err2_0M>>2) + (err2_1>>1) + (err2_2>>2);  // 2x 4x 2x

        temp  = temp * alpha3M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess1_1M;
        guess1_1M = max(0,min(temp,255));      // clip value

        temp   = (err2_0M>>2) + (err2_1>>1) + (err2_2>>2);  // 2x 4x 2x
        temp  += (err3_0M>>1) +  err3_1     + (err3_2>>1);  // 4x 8x 4x
        temp  += (err4_0M>>2) + (err4_1>>1) + (err4_2>>2);  // 2x 4x 2x

        temp  = temp * alpha3M;
        temp  = temp >> 7;          // numerator is assumed to be 4
        temp  = temp + guess3_1M;
        guess3_1M = max(0,min(temp,255));      // clip value
}

unsigned char rv = 0;

if( isOddPixel )
{
        rv = guess1_0M;
}
else
{
        temp = (old_err1_0M>>3)+(err1_0M>>2)+(err1_1>>3);//1x 2x 1x
```

```
        temp = temp>> 2; // divide by 4

        // make the signed value fit in one byte!
        // I could do a non-linear lookup table here!
        if( temp < -127 ) temp = -127;
        if( temp > 127  ) temp =  127;
        temp += 127;
        rv = (unsigned char)temp;
    }

    // Shift Error Values
    old_err1_0M = err1_0M;
    err1_0M = err1_1;
    err2_0M = err2_1;
    err3_0M = err3_1;
    err4_0M = err4_1;

    return rv; //return error value or subframe value
}
```